## Slide 1: Title

* Integrating Python with SQL (Part 2): Executing DML from Python — Dynamic Insertion into the Blog Database using sqlite3.[python](https://docs.python.org/3/library/sqlite3.html)AI-Enterprise-App-Development.pdf

## Slide 2: Today’s agenda

* Recap Part 1 (connect, cursor, execute, fetch) and extend to DML: INSERT, UPDATE, DELETE via Python’s sqlite3.AI-Enterprise-App-Development.pdf[python](https://docs.python.org/3/library/sqlite3.html)
* Design the user input flow for adding a post with safe parameterized queries and commits.[python](https://docs.python.org/3/library/sqlite3.html)AI-Enterprise-App-Development.pdf
* Class examples, hands‑on lab, real‑world safeguards, and bonus exercises for mastery.[sqlitetutorial](https://www.sqlitetutorial.net/sqlite-python/)AI-Enterprise-App-Development.pdf

## Slide 3: Where we are in Module 1

* This is Week 4 (Thu): “Integrating Python with SQL (Part 2) — Dynamic Insertion” per the official training plan.AI-Enterprise-App-Development.pdf
* The deliverable is a working Python script that takes user input and inserts a row into Posts in blog.db.[python](https://docs.python.org/3/library/sqlite3.html)AI-Enterprise-App-Development.pdf

## Slide 4: DML from Python — overview

* DML includes INSERT, UPDATE, DELETE; these modify data and must be followed by conn.commit() to persist changes in SQLite via sqlite3.[tutorialspoint+1](https://www.tutorialspoint.com/sqlite/sqlite_python.htm)
* Always use parameterized queries (placeholders) to prevent SQL injection and ensure proper value binding.[sqlitetutorial+1](https://www.sqlitetutorial.net/sqlite-python/)

## Slide 5: sqlite3 recap (DB‑API 2.0)

* sqlite3 is a standard library module that implements the Python DB‑API 2.0 interface for SQLite databases without external servers.[python+1](https://docs.python.org/3.9/library/sqlite3.html)
* Core flow: connect → cursor → execute (with parameters) → commit/rollback → close or use context manager for safer lifecycle.[python](https://docs.python.org/3/library/sqlite3.html)

## Slide 6: Blog DB assumptions

* Database: blog.db with tables Users(UserID, Username, Email, …), Posts(PostID, Title, Content, PublishedDate DEFAULT CURRENT\_TIMESTAMP, AuthorID FK), Comments(…) from prior Week 4 lectures.AI-Enterprise-App-Development.pdf
* Foreign key enforcement should be enabled when enforcing constraints, typically via PRAGMA foreign\_keys=ON at connection level if required by the design.[sqlite+1](https://sqlite.org/foreignkeys.html)

## Slide 7: Parameterized INSERT — why and how

* Use ? placeholders: cur.execute("INSERT INTO Posts (Title, Content, AuthorID) VALUES (?, ?, ?)", (title, content, author\_id)) to pass values safely.[tutorialspoint+1](https://www.tutorialspoint.com/sqlite/sqlite_python.htm)
* Never build SQL by string concatenation with user input; parameterization prevents injection and quoting errors.[sqlitetutorial+1](https://www.sqlitetutorial.net/sqlite-python/)

## Slide 8: Commit and transaction scope

* After successful DML, call conn.commit() to persist changes; uncommitted changes may be lost if the connection closes or an error occurs.[tutorialspoint+1](https://www.tutorialspoint.com/sqlite/sqlite_python.htm)
* Use with sqlite3.connect('blog.db') as conn: to automatically commit on success and rollback on exception for atomic operations.[mimo+1](https://mimo.org/glossary/python/sqlite)

## Slide 9: Validating foreign keys (AuthorID)

* Before insertion, verify that AuthorID exists in Users using SELECT 1 FROM Users WHERE UserID=? to provide clear feedback and avoid constraint errors.[sqlitetutorial+1](https://www.sqlitetutorial.net/sqlite-foreign-key/)
* If enforcing FK constraints, ensure PRAGMA foreign\_keys=ON is set per connection if constraints should be honored.[sqlite+1](https://www.sqlite.org/pragma.html)

## Slide 10: Class example — dynamic insertion (flow)

* Prompt for Title, Content, and AuthorID → validate inputs → verify AuthorID exists → perform parameterized INSERT → commit → report success.[sqlite+1](https://sqlite.org/foreignkeys.html)
* Handle exceptions and close the connection reliably to prevent resource leaks and partial writes.[python](https://docs.python.org/3/library/sqlite3.html)

## Slide 11: Class code — dynamic INSERT (complete)

pythonimport sqlite3  
  
def add\_post():  
 with sqlite3.connect("blog.db") as conn:  
 cur = conn.cursor()  
 # Optional: enforce foreign keys if schema depends on them  
 cur.execute("PRAGMA foreign\_keys = ON;")  
 title = input("Enter post title: ").strip()  
 content = input("Enter post content: ").strip()  
 author\_raw = input("Enter author ID: ").strip()  
 if not title or not content or not author\_raw.isdigit():  
 print("Invalid input."); return  
 author\_id = int(author\_raw)  
 cur.execute("SELECT 1 FROM Users WHERE UserID = ?", (author\_id,))  
 if cur.fetchone() is None:  
 print("Invalid AuthorID (no such user)."); return  
 cur.execute(  
 "INSERT INTO Posts (Title, Content, AuthorID) VALUES (?, ?, ?)",  
 (title, content, author\_id)  
 )  
 print("✅ Post inserted successfully.")  
if \_\_name\_\_ == "\_\_main\_\_":  
 add\_post()

* This uses a context manager for transactional safety and parameterized values for security per DB‑API guidance.[sqlite+1](https://www.sqlite.org/pragma.html)

## Slide 12: Explanation — code decisions

* Context manager ensures commit on success and rollback on exceptions, minimizing risk of inconsistent state.[python](https://docs.python.org/3/library/sqlite3.html)
* PRAGMA foreign\_keys=ON is used here when FK semantics are required by schema and business rules, as enforcement is per connection in SQLite.[sqlite+1](https://sqlite.org/forum/info/fd0b2d53bafc73f888069b3a0a3b15f35982c7e3fa910983b47db3e39ccabe18)

## Slide 13: Handling errors gracefully

* Catch sqlite3.Error to display a user‑friendly message and log the underlying exception for troubleshooting.[python](https://docs.python.org/3/library/sqlite3.html)
* Keep error handling around both validation and execute paths; commit should occur only when no exceptions are raised.[python](https://docs.python.org/3/library/sqlite3.html)

## Slide 14: Optional — timestamp handling

* Posts.PublishedDate can default to CURRENT\_TIMESTAMP in schema; alternatively, pass a timestamp from Python if needed for business logic.[sqlite](https://www.sqlite.org/docs.html)
* SQLite date/time functions (strftime, date, datetime) can also be used in queries for listing or filtering posts by recency.[sqlitetutorial+1](https://www.sqlitetutorial.net/sqlite-date-functions/sqlite-strftime-function/)

## Slide 15: UPDATE from Python — quick pattern

* To correct a post’s title: cur.execute("UPDATE Posts SET Title=? WHERE PostID=?", (new\_title, pid)) followed by commit.[python](https://docs.python.org/3/library/sqlite3.html)
* Always include a WHERE clause and validate the target row exists to avoid unintended mass updates.[tutorialspoint](https://www.tutorialspoint.com/sqlite/sqlite_python.htm)

## Slide 16: DELETE from Python — quick pattern

* To remove a post: cur.execute("DELETE FROM Posts WHERE PostID=?", (pid,)) followed by commit after confirmation.[python](https://docs.python.org/3/library/sqlite3.html)
* In business apps, consider soft delete via an IsActive flag to preserve history and support recovery.[sqlite](https://www.sqlite.org/docs.html)

## Slide 17: Class example — update and delete helpers

pythondef update\_title(conn):  
 cur = conn.cursor()  
 pid = input("PostID to update: ").strip()  
 if not pid.isdigit(): print("Invalid PostID."); return  
 new\_title = input("New title: ").strip()  
 cur.execute("UPDATE Posts SET Title=? WHERE PostID=?", (new\_title, int(pid)))  
 conn.commit(); print("Title updated.")  
  
def delete\_post(conn):  
 cur = conn.cursor()  
 pid = input("PostID to delete: ").strip()  
 if not pid.isdigit(): print("Invalid PostID."); return  
 cur.execute("SELECT Title FROM Posts WHERE PostID=?", (int(pid),))  
 row = cur.fetchone()  
 if not row: print("Not found."); return  
 ok = input(f"Delete '{row}' (y/n)? ").strip().lower()  
 if ok == 'y':  
 cur.execute("DELETE FROM Posts WHERE PostID=?", (int(pid),))  
 conn.commit(); print("Deleted.")

* These maintain parameterization and require explicit confirmation for destructive actions per best practice.[tutorialspoint+1](https://www.tutorialspoint.com/sqlite/sqlite_python.htm)

## Slide 18: Class example — list posts to verify

pythondef list\_posts(conn):  
 cur = conn.cursor()  
 cur.execute("""SELECT PostID, Title, PublishedDate, AuthorID  
 FROM Posts ORDER BY PublishedDate DESC""")  
 rows = cur.fetchall()  
 if not rows: print("No posts."); return  
 for r in rows:  
 print(f"[{r}] {r[1]} ({r[24]}) Author={r[25]}")

* Sorting by PublishedDate supports quick verification that newly inserted records are visible and correctly timestamped.[sqlite](https://www.sqlite.org/docs.html)

## Slide 19: Hands‑on lab — implement dynamic insertion

* Task: Write add\_post() exactly as shown, including validation, parameterized INSERT, commit, and success message.AI-Enterprise-App-Development.pdf[python](https://docs.python.org/3/library/sqlite3.html)
* Verify by listing posts in descending date and confirming the inserted row’s fields and author linkage.[sqlite](https://www.sqlite.org/docs.html)AI-Enterprise-App-Development.pdf

## Slide 20: Hands‑on lab — negative tests

* Try invalid AuthorID and ensure the script reports the problem without attempting the INSERT.[sqlite](https://sqlite.org/foreignkeys.html)
* Test empty title/content to confirm validation prevents bad writes and no commit occurs.[python](https://docs.python.org/3/library/sqlite3.html)

## Slide 21: Real‑world — integrity and FK enforcement

* If the system relies on referential integrity, enable PRAGMA foreign\_keys=ON per connection, as enforcement is not global nor on by default in all environments.[sqlite+1](https://www.sqlite.org/pragma.html)
* Validate relationships in code anyway to provide immediate, user‑friendly messages and avoid exposing low‑level constraint errors.[sqlite](https://sqlite.org/foreignkeys.html)

## Slide 22: Real‑world — transaction boundaries

* Prefer grouping logically related operations in a single transaction to ensure atomicity and easier rollback on composite errors.[python](https://docs.python.org/3/library/sqlite3.html)
* Avoid long‑running transactions that block writers; keep the unit of work small for responsiveness and reduced lock contention in SQLite.[sqlite](https://www.sqlite.org/docs.html)

## Slide 23: Real‑world — soft deletion option

* Add Posts.IsActive INTEGER DEFAULT 1; instead of DELETE, set IsActive=0 and filter IsActive=1 by default in listings to preserve audit history.[sqlite](https://www.sqlite.org/docs.html)
* This aligns with business analytics and regulatory needs where records should not be physically removed.[sqlite](https://www.sqlite.org/docs.html)

## Slide 24: Bonus — named parameters and executemany

* sqlite3 supports both positional (?) and named (:name) parameters for clarity in more complex statements with repeated values.[python](https://docs.python.org/3/library/sqlite3.html)
* executemany enables efficient batch inserts, e.g., bulk seeding posts from a prevalidated data source.[sqlitetutorial+1](https://www.sqlitetutorial.net/sqlite-python/)

## Slide 25: Bonus — input sanitation and length limits

* Sanitize inputs (strip whitespace, enforce max lengths) to protect data quality and avoid UI issues downstream.[python](https://docs.python.org/3/library/sqlite3.html)
* Consider rejecting unreasonably large Content blobs or storing long bodies in a separate table or file if needed by constraints or UI design.[sqlite](https://www.sqlite.org/docs.html)

## Slide 26: Bonus — date‑driven insertion and listing

* Let SQLite manage default timestamps via DEFAULT CURRENT\_TIMESTAMP for simpler insertion code paths.[sqlite](https://www.sqlite.org/docs.html)
* For listing by recency windows, use SQLite date functions like strftime/date/datetime to filter recent posts without extra Python logic.[sqlite+1](https://sqlite.org/lang_datefunc.html)

## Slide 27: Bonus — simple CLI wrapper

pythondef run():  
 with sqlite3.connect("blog.db") as conn:  
 while True:  
 print("\n1) Add 2) List 3) Update 4) Delete 5) Exit")  
 c = input("Choose: ").strip()  
 if c == '1': add\_post() # uses its own connection above; or refactor to pass conn  
 elif c == '2': list\_posts(conn)  
 elif c == '3': update\_title(conn)  
 elif c == '4': delete\_post(conn)  
 elif c == '5': break  
 else: print("Invalid.")

* A lightweight menu helps demo DML flows interactively before full CLI consolidation in the Friday project.AI-Enterprise-App-Development.pdf[python](https://docs.python.org/3/library/sqlite3.html)

## Slide 28: Debugging tips

* If INSERT appears to “fail,” confirm commit occurred, check table schema via PRAGMA table\_info(Posts), and verify the query in the sqlite CLI.[sqlite+1](https://sqlite.org/cli.html)
* Log exceptions (e.g., sqlite3.IntegrityError) to surface constraint problems early when testing validation paths.[python](https://docs.python.org/3/library/sqlite3.html)

## Slide 29: Testing checklist

* Positive: insert a valid post and confirm with SELECT; Negative: invalid AuthorID or empty fields block insertion; Edge: very long inputs or unicode characters.[python](https://docs.python.org/3/library/sqlite3.html)
* Re‑run after restarts to ensure persistence and correct commit behavior across application lifecycles.[python](https://docs.python.org/3/library/sqlite3.html)

## Slide 30: Security reminders

* Never interpolate user input into SQL strings; always use placeholders to avoid injection and quoting pitfalls.[sqlitetutorial+1](https://www.sqlitetutorial.net/sqlite-python/)
* Do not allow user‑controlled identifiers (table/column names); placeholders are only for values per DB‑API semantics.[python](https://docs.python.org/3/library/sqlite3.html)

## Slide 31: Performance hints

* Add indexes on common filters/joins (e.g., AuthorID, PublishedDate) to speed listings and author‑specific views as data grows.[sqlite](https://www.sqlite.org/docs.html)
* Use executemany for bulk inserts and keep transactions short to minimize lock durations in SQLite.[sqlite+1](https://www.sqlite.org/docs.html)

## Slide 32: Summary

* DML from Python requires parameterized queries, correct commit/rollback discipline, and input/relationship validation for safety and integrity.[sqlite+1](https://sqlite.org/foreignkeys.html)
* The dynamic insertion script is the foundation for interactive content creation and will evolve into a full CLI manager in the next session.AI-Enterprise-App-Development.pdf[python](https://docs.python.org/3/library/sqlite3.html)

## Slide 33: Hands‑on exercise (submit)

* Implement add\_post() with validation and PRAGMA foreign\_keys=ON; include a screenshot of a successful insert and a failed validation case.[sqlite+1](https://www.sqlite.org/pragma.html)
* Provide a short note describing how parameterization and commit were used to ensure safety and durability.[python](https://docs.python.org/3/library/sqlite3.html)

## Slide 34: Bonus exercises

* Add soft delete (IsActive) and modify listings to show only active posts; add a “restore” path to flip IsActive back to 1 when needed.[sqlite](https://www.sqlite.org/docs.html)
* Implement executemany to bulk‑insert 3 demo posts from a Python list while reusing validation logic per row.[sqlitetutorial+1](https://www.sqlitetutorial.net/sqlite-python/)

## Slide 35: References

* Python sqlite3 documentation (DB‑API 2.0): connect/cursor/execute/commit/rollback/parameters.[python+1](https://docs.python.org/3.9/library/sqlite3.html)
* SQLite documentation: SQL syntax, PRAGMA foreign\_keys, and date/time functions for strftime/date/datetime operations.[sqlite+2](https://sqlite.org/lang_datefunc.html)
* Course plan alignment: Week 4 Thu — Integrating Python with SQL (Part 2) Dynamic Insertion.